Lesson:

Complexity Analysis

Pre-Requisites:

- Basic Java Syntax
- if else statement
- Loops

We have discussed:

- Java basics
- If else statement
- Loops

Concepts Involved:

- Why is Complexity Analysis required ?
- Given two algorithms for a task, how do we find out which one is better?
- Time Complexity
- Need for Time Complexity
- Types of Time complexity Analysis
- Space Complexity
- Need for Space Complexity

In this lecture, we are going to study complexity analysis in a program. We will see why it is required and how we do it for different programs. We will discuss time complexity and space complexity and how they are important to consider for a given program. After that we will discuss analysis of recursion and how we can make a recursion tree. At last we will discuss space complexity in detail to know its importance and the ways to optimize it. So let's start with the lecture.

Why is Complexity Analysis required ?

Sometimes, there are more than one way to solve a problem. So, We need to learn how to compare the performance of different algorithms and choose the best one to solve a particular problem. While analyzing an algorithm, we mostly consider time complexity and space complexity. To know how efficient an algorithm or a program is, we need to do complexity analysis.

Given two algorithms for a task, how do we find out which one is better?

One naive way of doing this is - implement both the algorithms and run the two programs on your computer for different inputs and see which one takes less time.
There are many problems with this approach for analysis of algorithms.
1) It might be possible that for some inputs, the first algorithm performs better than the second. And for some inputs the second performs better.
2) It might also be possible that for some inputs, the first algorithm performs better on one machine and the second works better on other machine for some other inputs.

Time Complexity
● Applying the asymptotic analysis to measure the time requirement of an algorithm as a function of input sizes is known as time complexity. We assume each instruction takes a constant amount of time for time

complexity analysis.
● Total time complexity of a program is equal to the summation of all the running time of disconnected fragments.

Need for Time Complexity

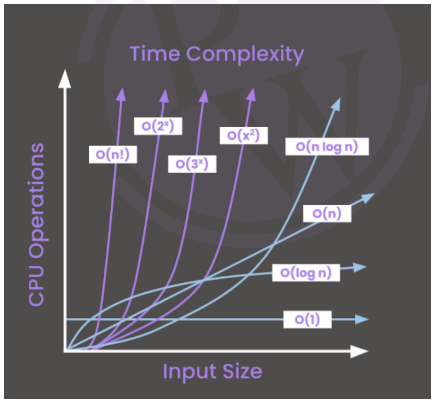
When analyzing any algorithm, we need to evaluate the effectiveness of that algorithm. Accordingly, we need to prefer the most optimized algorithm so as to save the time taken by it to execute. An example for the same could be linear search and binary search. Let's suppose we need to search a given value in a sorted array of size 8. This would take 8 iterations for linear search whereas it would just take log(8) ~ 3 iterations for binary search to search the same element, thereby saving a lot of time. Time Complexity for both binary search and linear search will be discussed in further lectures.

Types of Time Complexity Analysis:

a. Worst Case Time Complexity:
It is that case where the algorithm takes the longest time to complete its execution. It is represented by the Big-O notation. By default we calculate the worst case time complexity when we refer to the time complexity of a code.

b. Best Case Time Complexity: It is that case where the algorithm takes the least time to complete its execution. It is represented by the Omega Notation (Ω-Notation).

c. Average Case Time Complexity: As the name suggests, it gives average time for a program to complete its execution. It is represented by the theta notation (Θ-Notation).



The above graph represents how an increase in input size has an impact on the no. of CPU operations of a program for different time complexities of the program.

Constant Time Complexity	$0(1)$
Logarithmic Complexity	$0(\log_n)$
Linear Complexity	$0(n)$
Quadratic Complexity	$0(n^2)$
Cubic Complexity	$0(n^3)$
Polynomial complexity	$0(n^m), m>3$
Exponential Complexity	$0(a^n), a>1$

The above table shows different types of time complexities which are possible in a program. This list is not exhaustive and there are many more types of time complexities which are possible which is due to different parts of programs having different individual time complexities.

Let's now see some examples to understand how we calculate time complexity for different codes.

Example 1

Code:

```
for (int i = 0; i < n; i++) {
    // some operation of O(1) complexity
}
```

Explanation:
The above loop has a time complexity of O(n), where n is the number of iterations as the loop is running for n iterations.

Example 2

Code:

```
int i = 0;
while (i < n) {
    // some operation of O(1) complexity
    i++;
}
```

Explanation
The above while loop has a time complexity of O(n), where n is the number of iterations as the loop will run for n times.

Example 3

Code:

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        // some operation
    }
}
```

Explanation:
The above nested for loop has a time complexity of O(n^2), where n is the number of iterations of the outer loop. The reason for this complexity is that the inner loop is running for n times for each outer loop iteration making the whole time complexity quadratic.

Space Complexity

Space complexity is a measure of the amount of memory an algorithm uses, in terms of the sizes of the input. Like time complexity, it is often expressed using big O notation.

Need for Space Complexity

The amount of space a system has can be limited and therefore we need to optimize the memory/space taken by the algorithm to execute on that particular system with bounded space limits.

Let's understand space complexity with different examples:

Example 1:

Code:

```
void printHello() {
    String hello = "Hello, World!";
    System.out.println(hello);
}
```

Explanation:
The above code has constant space complexity because the amount of memory required by the code does not depend on the size of the input. So, This is O(1) space complexity code.
Example 2:

Code:

```
int[] copyArray(int[] arr) {
    int[] copy = new int[arr.length];
    for (int i = 0; i < arr.length; i++) {
        copy[i] = arr[i];
    }
    return copy;
```

Explanation:
The above code has O(n) space complexity i.e linear space complexity because we are creating an array of similar size as taken as a parameter in the function. So, more the length of the parameter array means more will be the length of the copy array resulting in direct relation. That's why O(n) space complexity.

Example 3:

Code:

```
int addUpto(int n){
    if (n ⩽ 0){
        return 0;
    }
    return n + addUpto(n-1);
}
```

Explanation:
Here each call add a level to the stack :

1. addUpto(4)
2. -> addUpto(3)
3. -> addUpto(2)
4. -> addUpto(1)
5. -> addUpto(0)

Each of these calls is added to the call stack and takes up actual memory.
So it takes O(n) space.

Time complexity of recurrence relations

Pre-requisites

- Recursion
- What is time complexity?

List of concepts involved

- Substitution method
- Recurrence tree method

Time complexity of recurrence relations:

A recurrence relation is a functional relation between the independent variable x, dependent variable f(x) and the differences of various order of f (x).
For example, we studied the fibonacci series where we have seen that any ith term is a function of previous two terms, i.e. (i-1)th term and (i-2)th term.

For any independent variable 'n', the corresponding fibonacci value is obtained by:
f(n) = f(n - 1) + f(n - 2) .....(1)
The equation (1) is known as recurrence relation.
Similarly, many algorithms are recursive. When we analyze them, we get a recurrence relation for time complexity. In order to find time complexity of such relations we will be discussing two methods in this class in great detail.

Substitution method:

The Substitution Method Consists of two main steps:
1. Guess the Solution.
2. Use principles of mathematical induction to find the boundary condition and show that our guess is correct.

Let's go with various examples to discuss more on the Substitution method, how it works and how to find time complexity using this method.

Question 1: Find the time complexity of the recurrence relation given by:

T(n) = 2T(n/2) + 4n

Solution: As discussed in the steps firstly we need to make a random guess towards our solution.

Let the solution be T(n) = O(nlogn).
The second step is to use mathematical induction to prove our guess to be correct.
So we have to prove that = T(n) <= c*n*logn, where c is a constant. We can assume that it is true for values smaller than n.
T(n) = 2T(n/2) + 4n
<= 2cn/2Log(n/2) + 4n
= cnLogn – cnLog2 + 4n
= cnLogn – cn + 4n
Ultimately T(n) <= cnLogn
Hence we can conclude the overall worst case time complexity to be O(nlogn).

Question 2: Determine the recurrence relation for the following series 1,7,31,127,511.
Solution : Let's analyze the sequence,
7 - 1 = 6
31 - 7 = 24 which can also be written as 4 * 6
127 - 31 = 96 which can be written as 4 * 24
511 - 127 = 372 which can be written as 4 *96
Now 2nd term 7 can be written as 4 * 1 + 3
Then 31 can be written as 4 * 7 + 3
. . . .and so on.
Hence the recurrence relation so forming is T(n) = 4 * (n - 1) + 3.

Recurrence tree method:

Recursion Tree is another method for solving the recurrence relations.
A recursion tree is a tree where each node represents the cost of a certain recursive sub-problem.
We sum up the values in each node to get the cost of the entire algorithm.

Steps to Solve Recurrence Relations Using Recursion Tree Method-

Step-01:
- Draw a recursion tree based on the given recurrence relation.

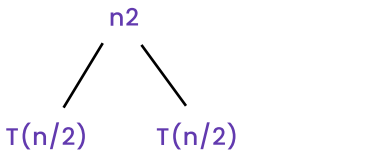
Step-02:
- Determine-
  - Cost of each level
  - Total number of levels in the recursion tree
  - Number of nodes in the last level
  - Cost of the last level

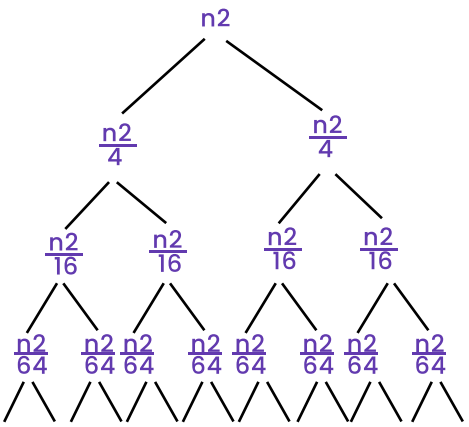
Step-03:
- Add the cost of all the levels of the recursion tree and simplify the expression so obtained in terms of asymptotic notation.

Question 1: Solve the following recurrence relation using recursion tree method-

T(n) = 2T(n/2) + n^2

Solution: The recurrence tree can be formed as:





@designer team please design these images as per PW standards.

T(n) = sum of time at each level
T(n) = n^2 + n^2/2 + n^2/4 + ... Log n (to the base 2) times.

Logn is the length/height of the recurrence tree so formed.

Total number of levels in the recursion tree-

- Size of sub-problem at level-0 = n/20
- Size of sub-problem at level-1 = n/21
- Size of sub-problem at level-2 = n/22

Continuing in similar manner, we have-

Size of sub-problem at level-i = n/2i
Suppose at level-x (last level), size of the sub-problem becomes 1. Then-
n / 2x = 1
2x = n
Taking log on both sides(with base 2), we get-
xlog2 = logn
x = log2n

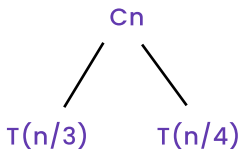
T(n) is an infinite GP with common ratio (r) = ½
Sum of infinite GP = a/(1-r) where a = first term and r = common ratio

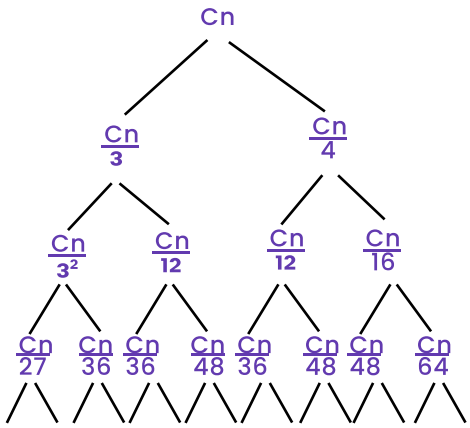
T(n) = n^2 / (1 - 1/2) = 2*n^2

Hence we can conclude the time to be T(n) = O(n^2) [considering upper bound for worst case complexity].

Question 2: Find the time complexity of the following recurrence relation using recurrence tree method.
T(n) = T(n/3) + T(n/4) + kn where k = constant

Solution:





To get total time sum values at each level of the recurrence tree:
T(n) = cn + cn/3 + cn/4 + cn/9 + cn/12 ..... Logn time.
The height of the tree is logn which is proved already in above examples as well.
T(n) = cn[1 + ⅓ + ¼ + 1/9 + 1/12+ ...]
T(n) = cn[1 + 7/12 + (7/12)^2 + . . .]
This is again sum of infinite GP with common ratio = 7/12
And first term = 1

Sum of infinite GP = a/(1 - r)
T(n) = cn[1/(1 - 7/12)] = cn*12/5

Therefore T(n) = O(n) . . . [considering upper bounds, ignoring constants]

Master Theorem

Pre-requisites:

- Recursion and recurrence relation
- What is time complexity?

List of concepts involved:

- Master theorem

Introduction:

The most widely used method to analyze or compare various algorithms is by computing their time complexities. As the algorithm gets complex, the time complexity function calculation also complexifies. Recursive functions call themselves in their body. It's more difficult if we start calculating its time complexity function by other commonly used simpler methods like substitution methods or recurrence tree methods. Master's theorem method is the most useful and easy method to compute the time complexity function of recurrence relations.

We can apply Master's Theorem for:

1. Dividing functions
2. Decreasing Functions

We will get into more detail what master theorem is and will see various examples to analyze the complexity of various functions.

Master theorem:

The master theorem is a method to calculate time complexities of recurrence relations of the following type:

$$T(n) = a\,T\left(\frac{n}{b}\right) + \theta\left(n^k \log^p n\right)$$

Master's Theorem

where n = size of the problem
a = number of subproblems in the recursion and a >= 1
n/b = size of each subproblem
b > 1, k >= 0 and p is a real number.
Then,

1. if $a > b^k$, then $T(n) = \theta(n^{\log_b a})$
2. if $a = b^k$, then
   (a) if $p > -1$, then $T(n) = \theta(n^{\log_b a} \log^{p+1} n)$
   (b) if $p = -1$, then $T(n) = \theta(n^{\log_b a} \log\log n)$
   (c) if $p < -1$, then $T(n) = \theta(n^{\log_b a})$
3. if $a < b^k$, then
   (a) if $p >= 0$, then $T(n) = \theta(n^k \log^p n)$
   (b) if $p < 0$, then $T(n) = \theta(n^k)$

Question 1: Given a recurrence relation:
T(n) = 3T(n/5) + 5n^2
Find the time complexity of this relation using the master theorem.

Solution:

From the given recurrence relation we can identify the variables a, b, p and k.
T(n) = 3T(n/5) + 5n^2

Here, a = 3 , b = 5 , k = 2 , p = 0
b^k = 5*5 = 25
a < b^k that corresponds to case 3.

Now since p >= 0
Therefore T(n) = θ(n^k * (logn)^p)
T(n) = θ(n^2) because (logn) ^ 0 = 1.

Question 2: Given a recurrence relation:
T(n) = 2T(2n/3) + 1
Find the time complexity of this relation using the master theorem.

Solution: Comparing the original equation of the master's theorem, the respective values of a ,b, p and k are 2 , 3/2, 0 , 0.

a > b^k because a = 2 and b^k = 3/2 ^ 0 = 1.
T(n) = θ(n ^ (logb a))
Therefore T(n) = θ(nlog1.5 2) [n raised to the power of log 2 at base 1.5]

Limitations of Master Theorem:

Relation function cannot be solved using Mater's Theorem if:

1. T(n) is a monotone function
2. a is not a constant
3. f(n) is not a polynomial

Examples where Master theorem can't be applied.

1.T(n) = cosx
2. T(n) = nT(n/3) + f(n)